```
executor.addCommand(right)
executor.addCommand(right)
executor.addCommand(left)

println("Executed left: $countLeft")
println("Executed right: $countRight")
executor.execute()
println("Executed left: $countLeft")
println("Executed right: $countRight")
```

The following result will be produced:

```
Executed left: 0
Executed right: 0
Executing: LEFT
Executing: LEFT
Executing: LEFT
Executing: RIGHT
Executing: RIGHT
Executing: LEFT
Executed left: 4
Executed right: 2
```

Interfaces

“Interfaces” are abstract types that are used to specify a behavior that classes must implement. Interfaces are declared using the ”interface” keyword, and may only contain function signature and constant declarations. “Interfaces” can contain declarations of abstract functions, as well as function implementations. What makes them different from abstract classes is that interfaces cannot store states. Interfaces can have properties. However, the properties need to be abstract or to provide accessor implementations.

Interfaces cannot be instantiated, but rather are implemented. A class that implements an interface must implement all of the non-default functions described in the interface, or be an abstract class.

We will demonstrate the usage of “interfaces” by a simple example. From book code examples open “Interfaces.kt”:

```
interface Vehicle {
    fun startEngine()

    fun stopEngine()

    fun drive()
}
```

We have defined a simple “interface” called “Vehicle”. Each vehicle can:

- start the engine
- stop the engine
- and drive.

Based on this we can define a class for “Car” as the type of vehicle:

```
open class Car : Vehicle {

    override fun startEngine() {

        println("start")
    }

    override fun stopEngine() {

        println("stop")
    }

    override fun drive() {

        println("drive")
    }
}
```

And finally, we will define a concrete car brand called ”Mercedes”:

```
class Mercedes(private val model: String) : Car() {

    override fun startEngine() {

        super.startEngine()
        println("$model: start")
    }

    override fun stopEngine() {

        super.stopEngine()
        println("$model: stop")
    }

    override fun drive() {

        super.drive()
        println("$model: drive")
    }
}
```

So, let’s play a bit with our Mercedeses:

```
val m1 = Mercedes("Mercedes-Benz AMG A 35")
val m2 = Mercedes("Mercedes-Benz AMG C 63")
val m3 = Mercedes("Mercedes-Benz AMG C 63 S")
val m4 = Mercedes("Mercedes-Benz CLA 250")

val scooter = object : Vehicle {

    override fun startEngine() {

        println("Scooter: start")
    }

    override fun stopEngine() {

        println("Scooter: stop")
    }

    override fun drive() {

        println("Scooter: drive")
    }
}

val vehicles = listOf(m1, m2, m3, m4, scooter)
```

```
vehicles.forEach {

    it.startEngine()
    it.drive()
    it.stopEngine()
    println()
}
```

As you can see we have defined four Mercedeses and a scooter, all of tehm represent “Vehicle”. If we run our program the following output will be produced:

```
start
Mercedes-Benz AMG A 35: start
drive
Mercedes-Benz AMG A 35: drive
stop
Mercedes-Benz AMG A 35: stop

start
Mercedes-Benz AMG C 63: start
drive
Mercedes-Benz AMG C 63: drive
stop
Mercedes-Benz AMG C 63: stop

start
Mercedes-Benz AMG C 63 S: start
drive
Mercedes-Benz AMG C 63 S: drive
stop
Mercedes-Benz AMG C 63 S: stop

start
Mercedes-Benz CLA 250: start
drive
Mercedes-Benz CLA 250: drive
stop
Mercedes-Benz CLA 250: stop

Scooter: start
Scooter: drive
```

```
Scooter: stop
```

Properties in Interfaces

In interfaces, properties can be declared. Property can be abstract or it can provide implementations for accessors. It is important to note that properties declared in interfaces can't have backing fields. Because of this accessors declared in interfaces can't reference them.

From book code examples open “InterfaceProperties.kt”:

```
interface Process {

    val cores: Int

    val memory: Int
        get() = 64

    fun execute()

    fun getUsedResources() = "cores=$cores, memory=$memory"
}
```

As you can see “Process” interface has two properties: “cores” and “memory”, both Integer data type. Let’s implement our interface:

```
class Download(val what: URL) : Process {

    override val cores: Int
        get() = 4

    override fun execute() {

        println(
            "Downloading: $what, using(${getUsedResources()})"
        )
    }
}

class Encrypt(val what: String) : Process {
    override val cores: Int
        get() = 8

    override val memory: Int
        get() = 128

    override fun execute() {

        println(
            "Encrypting: $what, using(${getUsedResources()})"
        )
    }
}
```

We have implemented the interface in two classes: “Download” and “Encrypt”. Overriding “cores” property is mandatory since the property is abstract. However, for the “memory” property we have a default value defined which means that we do not have to override it.

Interfaces Inheritance

In Kotlin interface can inherit from other interfaces. Because of this, both provide implementations and both declare new properties and functions. Classes that implement these interfaces are required to define the missing implementations. Let’s take a look at a proper example that demonstrates this. From book code examples open “InterfaceInheritance.kt”:

```
interface Device {

    val model: String
}

interface AudioDevice : Device {

    val brand: String
    val serialNumber: Long

    override val model: String
        get() = "$brand::$serialNumber"

    fun play()
}
```

```
class MusicPlayer(
    override val brand: String,
    override val serialNumber: Long
) : AudioDevice {

    override fun play() {

        println("'$model' is playing")
    }
}
```

“Device” is our first ”interface”. It is really simple. It has only one property: “model”. ”AudioDevice” device inherits “Device” interface, it overrides the “model” property and introduces two new properties and “play” function signature. “MusicPlayer” implements all this. It overrides properties in its constructor and provides “play” function implementation. Let’s instantiate it and try out instantiated objects:

```
listOf(

    MusicPlayer("Sony", 1241),
    MusicPlayer("Panasonic", 1001001),
    MusicPlayer("Sony", 1242)
).forEach {

    it.play()
}
```

Executing this will produce the following output:

```
'Sony::1241' is playing
'Panasonic::1001001' is playing
'Sony::1242' is playing
```

Overriding conflicts

If we inherit more than one implementation of the same function we conflict will occur. From book code examples open “Conflicts.kt”:

```
interface IDummy1 {
```